Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing

Hasan Ferit Eniser
MPI-SWS
Germany
hfeniser@mpi-sws.org

Timo P. Gros
Saarland University, Saarland
Informatics Campus
Germany
timopgros@cs.uni-saarland.de

Valentin Wüstholz
ConsenSys
Germany
valentin.wustholz@consensys.net

Jörg Hoffmann
Saarland University, Saarland
Informatics Campus
German Research Center for Artificial
Intelligence (DFKI)
Germany
hoffmann@cs.uni-saarland.de

Maria Christakis
MPI-SWS
Germany
maria@mpi-sws.org

ABSTRACT

Testing is a promising way to gain trust in a learned action policy π, in particular if π is a neural network. A "bug" in this context constitutes undesirable or fatal policy behavior, e.g., satisfying a failure condition. But how do we distinguish whether such behavior is due to bad policy decisions, or whether it is actually unavoidable under the given circumstances? This requires knowledge about optimal solutions, which defeats the scalability of testing. Related problems occur in software testing when the correct program output is not known.

Metamorphic testing addresses this issue through metamorphic relations, specifying how a given change to the input should affect the output, thus providing an oracle for the correct output. Yet, how do we obtain such metamorphic relations for action policies? Here, we show that the well explored concept of relaxations in the Artificial Intelligence community can serve this purpose. In particular, if state s' is a relaxation of state s, i.e., s' is easier to solve than s, and π fails on easier s' but does not fail on harder s, then we know that π contains a bug manifested on s'.

We contribute the first exploration of this idea in the context of failure testing of neural network policies π learned by reinforcement learning in simulated environments. We design fuzzing strategies for test-case generation as well as metamorphic oracles leveraging simple, manually designed relaxations. In experiments on three single-agent games, our technology is able to effectively identify true bugs, i.e., avoidable failures of π, which has not been possible until now.

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534392

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Reinforcement learning*.

KEYWORDS

metamorphic testing, fuzzing, action policies

ACM Reference Format:
Hasan Ferit Eniser, Timo P. Gros, Valentin Wüstholz, Jörg Hoffmann, and Maria Christakis. 2022. Metamorphic Relations via Relaxations: An Approach to Obtain Oracles for Action-Policy Testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534392

1 INTRODUCTION

Action policies represented by neural networks are highly successful in complex, sequential decision making problems, specifically in games [32, 38, 39] and increasingly in Artificial Intelligence (AI) Planning [16, 19, 24, 25, 43]. Once a policy π has been learned, it can be used to make real-time decisions in dynamic environments, simply by calling $\pi(s)$ on the current state s to obtain the next action. This approach, however, comes with obvious safety concerns due to potential policy bugs, i.e., undesirable or even fatal policy behavior. Testing is a natural paradigm, given its scalability, to address these concerns.

But what is a "bug" in this context? In many environments, undesirable or fatal behavior can be unavoidable – e.g., traffic making it impossible to avoid a crash in autonomous driving, or a state in which it is impossible for a bipedal robot to keep its balance. Such situations are not bugs in π as the bad behavior is not actually due to bad policy decisions. In general, in order to know whether a situation constitutes a bug in π, we need to know the optimal policies, which minimize the probability of failure. This would defeat the scalability of testing.

Prior work on testing in the sequential decision making domain does not address this issue. It considers a "system" that takes decisions in an environment and tries to find situations where a failure condition ϕ is satisfied (e.g., [3, 12, 14, 27, 28], see [9] for a recent overview). Such an approach implicitly assumes that a correctly designed system – in our case, a learned action policy – can always avoid ϕ. This shortcoming was recently pointed out by Steinmetz et al. [40], who analyzed the possibility of identifying sub-optimal policy behavior through upper- and lower-bounding techniques. Here, we instead take inspiration from software testing, providing an alternative technique to detect avoidable failures.

Not knowing the optimal policies is akin to not knowing the correct output of a program. Metamorphic testing [8] addresses the latter by testing program behavior on inputs chosen such that it is known how the respective outputs should relate. That is, one specifies a *metamorphic relation*, encompassing a relation R^I over inputs together with a corresponding necessary relation R^O over outputs. If, for inputs i, i' with $R^I(i, i')$, the outputs o, o' do not satisfy $R^O(o, o')$, then we know there is a bug. Thus, R^O provides a test *oracle* for the correct output.

However, this oracle still requires knowledge about correct outputs in the form of R^O. For example, action decisions in autonomous driving can be tested using metamorphic relations derived from well known human-designed rules, such as "speed down by 25% if rainy" [10, 42, 45]. But how do we come by metamorphic relations in general, sequential decision making problems?

Our approach. We answer this question in terms of a relation R^O, not over specific output actions a vs. a' of a policy, but over *the space of solutions below states s vs. s'*. Such relations are very common and well explored in AI, namely in the form of over-approximations obtained through *relaxation*. More specifically, assume that states s and s' are related in terms of a relaxation relation $R(s, s')$, identifying that s' is easier to solve than s. If π fails on easier s' but not on harder s, then we know that π contains a bug manifested on s'. This is our key insight: *relaxations provide a means to specify metamorphic relations, and thus, a test oracle in general, sequential decision making problems.* Furthermore, this approach captures sequential policy behavior, rather than merely immediate outputs as in all other works on metamorphic testing.

As indicated by our notation above, we focus on *state relaxations* R, which modify only the state and not any other aspects of the agent's task. Such relaxations are often quite natural and easy to obtain. For example, when obstacles need to be avoided, states can be relaxed by removing obstacles; when resources are limited, relaxations can increase resource availability; when there are time constraints, these constraints can be relaxed (e.g., by postponing a deadline).

In this paper, we do not yet investigate the automatic generation of such relaxations. Instead, we perform case studies in three 2D-world single-agent games involving (fixed or moving) obstacles, and manually design relaxation relations $R(s, s')$, where s' has easier-to-avoid obstacles. It is important to note that, while this involves manual per-domain labor, it requires hardly any domain *knowledge* – relaxing obstacles is trivial. This is in stark contrast to knowing the difference between the optimal solutions for s and s', which constitutes the kind of knowledge we would need in order to design a traditional metamorphic output relation R^O.

Nevertheless, the automated design of relaxations in our context, including ones going beyond state relaxations, remains of course an important topic for future work. For this, there is huge potential to draw on the literature on relaxation design for heuristic functions i.e., to compute lower bounds on goal distance (e.g., [6, 11, 13, 21]). This is non-trivial though for a variety of reasons. In particular, rather than relaxing radically to obtain an efficiently computable heuristic function, we need to relax cautiously, in small steps, to be able to identify bugs.

Our testing framework addresses Markov decision processes (MDPs), of which we require access only to a simulator (given state s and action a, output an outcome state s'). For test-state generation, we take inspiration from fuzzing [1, 2, 30], which mutates program inputs randomly with a bias to maximize diversity. We transfer this idea to our setting by taking input mutations to be random action applications, and measuring test-state diversity in terms of Euclidean distance.

We implemented our techniques in a framework we call π-fuzz. We evaluate π-fuzz on three single-agent games, with policies learned by reinforcement learning. Our experiments show that fuzzing is effective in generating a diverse set of states, and that our metamorphic oracles are able to identify thousands of unique bugs even in well trained policies. (The source code of π-fuzz and all data necessary to reproduce our experiments are available at https://github.com/Practical-Formal-Methods/pi-fuzz.)

Contributions. In short, we make the following contributions:

- We introduce the first technique for metamorphic testing of action policies. Our key insight is to use the concept of relaxations to design novel oracles that do not require optimal policies.
- We propose the π-fuzz fuzzing framework, which combines a fuzzer for action policies with metamorphic test oracles.
- We evaluate π-fuzz on three single-agent games, with policies learned by reinforcement learning.

Outline. The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 provides background on the general setting and introduces notation. Section 4 gives an overview of π-fuzz. In Section 5, we formally define "policy bugs", and in Section 6, we describe our metamorphic test oracles. In Section 7, we describe our fuzzing component before introducing our three case studies in Section 8. Finally, we evaluate π-fuzz in Section 9 and conclude in Section 10.

2 RELATED WORK

Testing neural networks. Testing neural networks is a well studied field in the literature. However, most of this work focuses only on a single invocation of a neural network, for example, in the case of image classifiers. For instance, there are several techniques that introduce test coverage criteria at the neural-network level [18, 26, 29, 35] for more efficient testing. Sun et. al. [41] devise a technique for concolic testing of neural networks. Wang et. al. [44] detect adversarial inputs by using ideas from mutation testing. Finally, there are some existing testing techniques that use metamorphic relations in specific domains, such as automated driving [10, 42, 45], object detection [46], and translation [20].

However, applying such existing work to test reinforcement learning (RL) policies is non-trivial since such policies are typically assessed by the total reward at the end of a sequence of decisions (i.e., *multiple* network invocations) that are taken from a given initial state. Differentially testing the optimality of each decision in the sequence is impractical in many real-world domains since it would require access to an optimal policy. Apart from that, the aforementioned approaches for metamorphic testing of autonomous driving decisions [10, 42, 45] are domain specific and leverage human knowledge in the form of driving rules – knowledge about solutions that is not available in general.

On the other hand, our approach only relies on relaxation relations, which are typically straightforward and apply to a wide range of domains. To nevertheless provide an empirical comparison to rule-based approaches, we experiment with simple "change nothing" rules. These define metamorphic relations prescribing that, if s' is a relaxation of s, then what works for harder s should also work for easier s' – the policy should not change. However, this may inaccurately classify s' as a bug, leading to false positives (lack of *precision*). Further, our experiments show that this method has lower *recall* (more false negatives) than ours.

Bug finding for decision making policies. There is recent work on finding falsifying inputs for RL-based decision making policies in hybrid systems [3, 9, 12, 14, 27, 28]. Unlike π-fuzz, these techniques do not address scenarios where the policy is expected to fail (e.g., unavoidable crash state). As mentioned earlier, Steinmetz et al. [40] recently pointed this out and used upper- and lower-bounding techniques to identify sub-optimal policy behavior. In contrast, our work takes inspiration from metamorphic testing to ensure that the identified failures are avoidable.

A recent study [34] focuses on testing deep NN models solving MDPs via techniques such as reinforcement learning and imitation learning [22]. They develop a fuzzer that is guided by a heuristic novelty measure to detect crash triggering initial states. There are three key differences with our work: (1) Their state mutations employ hand-crafted bounds to suppress bugs due to unavoidable crash states. Configuring these bounds can be difficult, if not impossible, for complex domains. In contrast, our approach by construction never reports bugs due to unavoidable crashes. (2) Bounds that limit the magnitude of each mutation may result in insufficient exploration of the input space. In π-fuzz, such bounds are not necessary. (3) The aforementioned work can identify only crash-triggering states. In contrast, our approach can easily be generalized to identify states on which the reward is less than expected.

Safe reinforcement learning. There exists a large body of work on safe reinforcement learning [4, 5, 23]. On a high level, safe reinforcement learning refers to training RL agents that are guaranteed to satisfy given specifications (see Garcia et al. [15] for an overview). While safe RL and policy testing share the same goals, their methods are fundamentally different. However, we can imagine interesting combinations as part of our future work; for instance, by feeding detected bugs back into policy re-training, testing may become part of a larger safe-RL loop.

3 CONTEXT AND NOTATIONS

Our methods address discrete-time Markov decision processes as follows. An MDP is a tuple $M = (S, A, T, S_0)$ of **states** S; **actions** A; **transition probability function** $T : S \times A \mapsto \mathcal{D}(S)$, where $\mathcal{D}(S)$ denotes the set of probability distributions over S; and **initial states** $S_0 \subseteq S$ (of which one $s_0 \in S_0$ will be chosen randomly at execution time).

A **policy**, also **agent**, is a function $\pi : S \mapsto A$ that chooses actions in S. We consider policies π represented by neural networks. The policy is typically trained to maximize rewards associated with states or state transitions. Our approach is agnostic to how this is done. A **run** of a policy π on an (arbitrary) state $s_0 \in S$ is a state/action sequence $\sigma = \langle s_0, a_0, s_1, a_1, \dots \rangle$, where, for all i, $a_i = \pi(s_i)$ and $\mu(s_{i+1}) > 0$, where $\mu = T(s_i, a_i)$ and $\mu(s_{i+1})$ denotes the probability of reaching state s_{i+1} from s_i by taking action a_i.

Note that this definition of policy is restricted in terms of being memoryless and deterministic. Both restrictions can be lifted in principle, but deterministic memoryless policies are relevant in their own right and form a natural starting point for the investigation of metamorphic action-policy testing. We assume that π is represented as an NN classifier whose final layer can also be interpreted as a probability distribution $\hat{\pi}(s) \in \mathcal{D}(A)$ over actions. We make use of the latter in fuzzing, by sampling $\hat{\pi}(s)$ in order to explore MDPs in which random actions do not lead to interesting states.

We assume a given non-temporal **failure condition** ϕ that should be avoided by the agent – exploring our approach for temporal ϕ remains a topic for future work. We say that a run σ **fails** if there exists s_i along σ such that $s_i \models \phi$; otherwise, we say that σ **succeeds**. We denote by $P_\phi(\pi, s)$ the probability that the run of π on s fails, and by $P^*_\phi(s)$ the minimal such probability achieved by any policy.

We do not assume that we have a declarative model of M; a simulator suffices to apply our methods. We merely assume that the representation of states is state-variable based, i.e., each s is uniquely identified by a value assignment to a vector of **state variables** $(v_1, \dots, v_n)$. The domains of the state variables do not matter to our approach as long as Euclidean distance can be defined (needed in our state-diversity notion). For simplicity, we assume that the state variables are real-valued in this paper, i.e., states s map each v_i to $\mathbb{R}$.

We refer to the simulator as the **environment**, denoted E. It provides the programmatic interfaces E.**randomInit**(), which returns a random initial state $s_0 \in S_0$; E.**setState**(s), which sets the environment state to $s \in S$; and E.**step**(s, a), which, given $s \in S$ and $a \in A$, picks and outputs a state s' according to the distribution $T(s, a)$.

We furthermore assume that E has a parameter ρ – the random seed – and an interface E.**setSeed**(r) setting $\rho := r$. We use this interface in part of our methodology to fix specific environment behaviors and identify bugs pertaining to those. Namely, whenever we check whether π contains a bug manifested below a state s, we call E.setSeed(r) directly before running π on s, determinizing T as a function of state, action, and run length so far. We denote the resulting unique **run of** π **on** s **given random seed** r by $E^r.\sigma[\pi, s]$.

4 π-FUZZ POLICY FUZZING FRAMEWORK

Figure 1 provides a high-level overview of our π-fuzz policy-testing framework. As shown in the figure, π-fuzz takes as input a policy under test π and an environment E. The framework consists of two

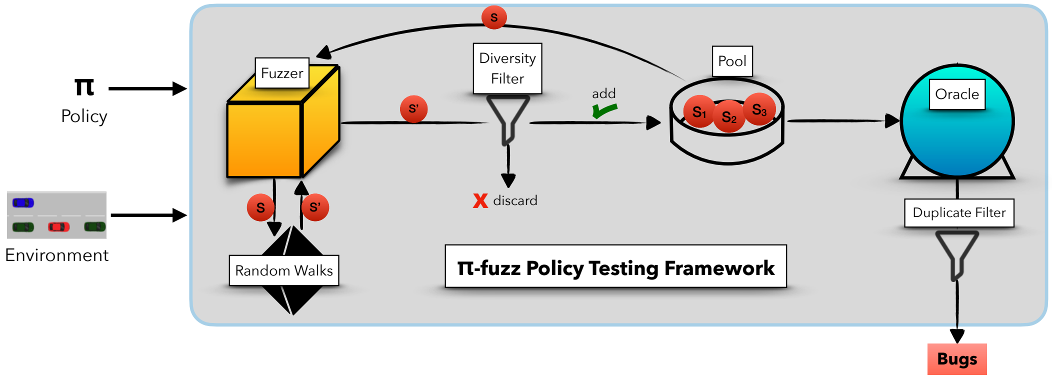


Figure 1: Overview of π-fuzz framework.

main components: the **fuzzer**, which generates a diverse pool of test states s_i; and the **oracle**, which identifies policy bugs among these test states. The fuzzer uses random walks to generate new states, which are then filtered by diversity to obtain the pool. We will describe our fuzzing algorithm in detail in Section 7. Our key contribution is the design of the oracle, via metamorphic relations based on relaxations. A duplicate filter at the end of this pipeline serves to provide unique bugs as the output of π-fuzz.

Our π-fuzz framework is implemented as a generic policy tester, independent of any specific environment (in fact, we use the same implementation across our case studies in this paper). The input interface for π-fuzz consists of the neural network representation of π, in the PyTorch format; the aforementioned programmatic interface of E implementing E.randomInit(), E.setState(s), E.step(s, a) and E.setSeed(r); as well as a programmatic interface for metamorphic operations underlying the oracle, explained in Section 6.

5 POLICY BUGS

In our context, we consider two notions of "policy bugs", one of which is specific to a fixed environment behavior, while the other quantifies over all possible such behaviors.

Definition 1 (Bug). Let $M = (S, A, T)$ be an MDP, E a simulator for M, π a policy, and $s \in S$ a state.

(i) We say that s is a **bug** in π if $P_\phi(\pi, s) > P^*_\phi(s)$.
(ii) Given a random seed $\rho = r$, we say that the run $E^r.\sigma[\pi, s]$ is a **seed-bug** in π if $E^r.\sigma[\pi, s]$ fails, but there exists a policy π' such that $E^r.\sigma[\pi', s]$ succeeds.

Bugs (i) arguably capture the canonical understanding of policy bugs when testing for failure avoidance in a probabilistic environment. Seed-bugs (ii) are an approximation that allows to consider individual environment behaviors. If π fails but π' succeeds given the same random seed, then this indicates that π is faulty. This is, however, not necessarily the case: (ii) does *not* in general imply (i) because the decisions causing failure on r may be beneficial for other environment behaviors. Hence, (ii) is merely a pragmatical proxy for (i). That said, (i) and (ii) coincide for deterministic environments; and in our case studies, most states s with seed-bugs found by our metamorphic oracles are in fact bugs. Moreover, (ii) is much faster to evaluate than (i), which makes it useful for practical purposes.

Note that seed-bugs are best characterized by the actual run $E^r.\sigma[\pi, s]$, rather than state s alone, as there can be many different environment behaviors below s and only some of them may exhibit the observed failure.

6 METAMORPHIC ORACLES

In this section, we explain the principle of relaxation-based metamorphic oracles, specify the oracles used in our case studies, and discuss how suitable relaxations may be obtained in general, especially when considering the relaxation literature.

6.1 Metamorphic Oracles via Relaxation

Obviously, Definition 1 cannot be tested efficiently on large state spaces. We adapt the idea of metamorphic testing to solve this issue. The key element is a state relaxation:

Definition 2 (State Relaxation). Let $M = (S, A, T)$ be an MDP, and E a simulator for M. We say that $t \in S$ **relaxes** $s \in S$ if, for every policy π_s, there exists a policy π_t such that, for every random seed $\rho = r$, whenever $E^r.\sigma[\pi_s, s]$ succeeds, then $E^r.\sigma[\pi_t, t]$ also succeeds.

We say that $R \subseteq S \times S$ is a (state) **relaxation** if, for every $(s, t) \in R$, t relaxes s.

We will illustrate and discuss this definition below. In a nutshell, a relaxed state t allows to adapt any policy for s to achieve the same (or more) failure-avoidance ability. In that sense, intuitively, "t is easier to solve than s". Definition 2 captures the most general condition under which this is the case, and where our metamorphic oracles hence work as intended.

Namely, the idea is quite simple: if t is easier to solve than s, but the policy π is worse on t than on s, then π's behavior on t must be wrong:

Proposition 3 (Metamorphic Oracle). Let $M = (S, A, T)$ be an MDP, E a simulator for M, and R a relaxation. Let $s, t \in S$ be states such that $(s, t) \in R$. We have:

(i) If $P_\phi(\pi, s) < P_\phi(\pi, t)$, then t is a bug in π.

(ii) If $E^r.\sigma[\pi, s]$ succeeds but $E^r.\sigma[\pi, t]$ fails, then $E^r.\sigma[\pi, t]$ is a seed-bug in π.

Proof. (i): First, note that we have $P^*_\phi(s) \geq P^*_\phi(t)$: Given a policy π^*_s that minimizes the probability of failure on s, by Definition 2 there exists a policy π_t for t that succeeds in all cases where π^*_s does. Hence, $P_\phi(\pi^*_s, s) \geq P_\phi(\pi_t, t)$, showing that $P_\phi(\pi^*_s, s) \geq P_\phi(\pi^*_t, t)$ for any policy π^*_t that minimizes the probability of failure on t. Together with prerequisite (i), we get $P_\phi(\pi, t) > P_\phi(\pi, s) \geq P^*_\phi(s) \geq P^*_\phi(t)$, which shows the claim.

(ii): By prerequisite, the run of π on s given random seed r succeeds, and $(s, t) \in R$. Hence, by Definition 2, there exists a policy π_t for which the run on t given r succeeds. As the latter is not the case for π, $E^r.\sigma[\pi, t]$ is a seed-bug in π. □

To illustrate Definition 2 and the kind of practical relaxations we employ in our experiments, let us briefly consider the case studies we contribute. We experiment with three games (Highway, LunarLander, BipedalWalker) where an agent moves in a 2D-world and needs to reach a target position while avoiding (fixed or moving) obstacles. In each case, our relaxation relation R modifies the game landscape, making obstacles easier to avoid. Figure 2 (see Section 8) illustrates this. For each of the three games, the figure shows a state s on the left and a relaxed state t on the right.

Highway involves a car (red car in Figure 2a) navigating traffic on a 2-lane highway. Less traffic is easier to navigate. In the sense of Definition 2, whenever π_s manages to avoid crashing into traffic when started from s, we can achieve the same when starting from t simply by taking the same driving decisions; i.e., the desired policy π_t behaves like π_s on the corresponding states. If the policy π under test takes different decisions on t, which crash more frequently, then π exhibits a bug on t. The other two games are similar in this regard: whenever π_s manages to avoid crashing on s, the same decisions avoid a crash on t, and so t relaxes s according to Definition 2.

A remarkable special case of Definition 2 is that of deterministic transitions, like in BipedalWalker where there is no noise in the effect of the robot's motor commands. In this case, the run of a policy from a state is unique, and t relaxes s if and only if either t is solvable (a succeeding policy exists), or s is unsolvable; in particular, all solvable states relax each other. This is very generous from a formal point of view, but it still makes perfect sense for bug detection: if s is solvable and $R(s, t)$, then we know that t is solvable. This is precisely the most general condition under which we can detect avoidable failures by comparing the behavior of π across states s, t: if $R(s, t)$ and π succeeds on s, then t is solvable, so failure of π on t constitutes a bug. Practical relaxation methods will, of course, instantiate the broad frame of Definition 2 with much more restrictive relations over states, like the relaxations in our case studies.

6.2 Metamorphic Oracles in our Case Studies

Given a state relaxation R as per Definition 2, Proposition 3 provides a tool to detect bugs and seed-bugs. We turn this into practical oracles for π-fuzz by sampling R a given number of times. Algorithm 1 specifies three different oracles along these lines. Let us discuss them from top to bottom.

Algorithm 1: Metamorphic oracles

```
Function BugOracle($R, \pi, s_i$):
    evaluate $P_\phi(\pi, s_i)$;
    repeat ORACLE_BUDGET times
        $t_i$ = RandomState($\{t_i \mid R(t_i, s_i)\}$);
        evaluate $P_\phi(\pi, t_i)$;
        if $P_\phi(\pi, t_i) < P_\phi(\pi, s_i)$ then
            return 1
    return 0;
Function BasicSeedBugOracle($R, \pi, s_i, r$):
    if $E^r.\sigma[\pi, s_i]$ fails then
        repeat ORACLE_BUDGET times
            $t_i$ = RandomState($\{t_i \mid R(t_i, s_i)\}$);
            if $E^r.\sigma[\pi, t_i]$ succeeds then
                return 1
    return 0;
Function ExtSeedBugOracle($R, \pi, s_i, r$):
    $B = \{\}$;
    if $E^r.\sigma[\pi, s_i]$ succeeds then
        repeat ORACLE_BUDGET times
            $t_i$ = RandomState($\{t_i \mid R(s_i, t_i)\}$);
            if $E^r.\sigma[\pi, t_i]$ fails then
                $B = B \cup \{E^r.\sigma[\pi, t_i]\}$;
    return $B$;
```

The BugOracle algorithm checks whether a given test-pool state s_i generated by π-fuzz can be identified to be a bug. It does so by comparing $P_\phi(\pi, s_i)$ with $P_\phi(\pi, t_i)$ for *unrelaxed* states t_i, i.e., harder states where $R(t_i, s_i)$. By Proposition 3, if the Boolean return value is 1, then s_i is a bug in π. Evaluating P_ϕ here is a sub-problem, and solving it precisely is intractable in itself for large state spaces. In our implementation, we approximate P_ϕ by sample runs.

The BasicSeedBugOracle algorithm proceeds in a similar manner, but checks for seed-bugs instead. The random seed r is an input to the oracle (set by the fuzzer, see Section 7) as the oracle's job is to identify bugs given a fixed environment behavior. The oracle returns 1 if π fails on the test state s_i but succeeds on one of the unrelaxed states t_i. By Proposition 3, $E^r.\sigma[\pi, s_i]$ is a seed-bug in this case.

Finally, ExtSeedBugOracle is an extension that applies in case π succeeds on the test state s_i given r. In this case, $E^r.\sigma[\pi, s_i]$ cannot be a seed-bug, but we may be able to identify relaxed states t_i as seed-bugs instead. The oracle leverages this possibility in the obvious manner. In our case studies, many additional seed-bugs are found in this way. Note that we can define a similar extended version of BugOracle; however, such an oracle would be very slow.

To sample the relaxation relation R, our implementation in π-fuzz assumes that R is given in the form of a set of **metamorphic operations**: state-modification operators that either relax the given state (e.g., by removing obstacles) or unrelax it (e.g., by adding obstacles). The sampling then simply consists in applying a randomly chosen metamorphic operation with randomly chosen parameters (e.g., which obstacles to remove, or where to add new obstacles).

Importantly, the magnitude of metamorphic operations affects oracle efficacy. If π works well on s_i and t_i is much easier, it is unlikely that the policy is bad on t_i, thus not leading to the detection of a bug. If π is bad on s_i and t_i is much harder than s_i, it is unlikely that the policy works well on t_i, again not leading to the detection of a bug. Therefore, in both directions, metamorphic operations should be applied cautiously, making small modifications only. Our operations modifying individual state attributes naturally support this. Also, for this reason, we do not chain over metamorphic operations, always applying only a single such operation when sampling R in Algorithm 1.

Section 8 outlines the metamorphic operations we use in each of our case studies. The algorithm parameter `ORACLE_BUDGET` is set to 500 in all our experiments.

6.3 Discussion: How To Obtain Relaxations

How can state relaxations for metamorphic oracles be obtained? In some special cases, relaxations modifying individual state attributes are actually quite easy to come by. Wherever obstacle avoidance plays a role, we can define metamorphic operations making obstacles easier to avoid, as in our case studies. Other simple examples include MDPs where resources, like fuel or energy, are consumed: we can then relax states by increasing the amounts of resources available. Similarly to this, if the MDP involves (discrete-time) deadlines by which something needs to be achieved, then states can be relaxed by postponing the deadlines. More generally, if some actions are possible only within given discrete-time time windows, then we can relax states by broadening those time windows. Note that such resource and time-constraint relaxations can potentially even be obtained fully automatically, as the metamorphic operations needed are generic (add resource/expand time-window border).

It remains, of course, an important question whether and how we can tap into the potential of research on relaxations in AI, where relaxations have been intensively investigated for the design of heuristic functions, i.e., to compute lower bounds on goal distance (e.g., [6, 11, 13, 21]). The adaptation of these methods to our context is highly non-trivial as relaxations underlying heuristic functions make strong problem simplifications (e.g., abstractions over-approximating transition behavior). This is in contrast to our need for cautious relaxation in small steps, as outlined above.

Another, perhaps more promising, source of state relaxations can be simulation relations [17, 31], where $R(s, t)$ holds – t simulates s – iff for every outgoing transition of s there is a corresponding outgoing transition in t, leading to simulating outcome states $R(s', t')$. Such relations can potentially be extracted automatically if a declarative model of the MDP is available.

7 FUZZING ALGORITHM

We now discuss the fuzzer component from Figure 1 in more detail. The fuzzer builds up a pool of diverse states by relying on two subcomponents, namely random walks and diversity analysis. Consider the pseudo-code in Algorithm 2.

Algorithm 2: Fuzzing procedure

```
1  Function Fuzzer(Env E, Policy π):
2      P = [];
3      P = add(E.randomInit(), P);
4      while ¬interrupted() do
5          if randomBoolean(INC_PROB) then
6              s = randomState(P);
7          else
8              s = E.randomInit();
9          s′ = randomWalk(E, π, s);
10         if $\min_{t \in P} d^{\mathrm{Eucl}}(s', t) >$ DIV_THRESH then
11             P = add(s′, P);
12     for each $s_i \in P$ do
13         Run oracle on $s_i$ (picking r if needed);
14 Function randomWalk(Env E, Policy π, State s):
15     E.setState(s);
16     k = randomIntRange(0, WALK_LENGTH);
17     if randomBoolean(POL_PROB) then
18         repeat k times
19             a = randomPolicyAction($\hat{\pi}$(s));
20             s = E.step(s, a);
21     else
22         repeat k times
23             a = randomAction(E.actions);
24             s = E.step(s, a);
25     return s;
```

First, the fuzzer adds a random initial state to the pool of states P (line 3). Until the fuzzer is interrupted (e.g., via a user-provided time limit), it tries to incrementally expand P. To do so, it randomly decides (biased by a probability provided in parameter `INC_PROB` on line 5) to either select a random state from the pool (line 6) or select a new random initial state (line 8). Utilizing previously generated states (from the pool) allows us to explore the state space much more effectively as they are used as stepping stones to delve into unexplored territory. A random walk is then conducted on the resulting state s to obtain a new candidate state s' (line 9). If s' is sufficiently diverse, it is added to P (lines 10 – 11). Here, $d^{\mathrm{Eucl}}(s, s')$ is the Euclidean distance between s and s', and `DIV_THRESH` sets a threshold for the minimum distance to the states already in pool P.

Once pool P is final, an oracle is called on each state $s_i \in P$ (lines 12 – 13). If the oracle requires a fixed random seed – like the two seed-bug oracles from Algorithm 1 – then the fuzzer chooses that seed here. Note that, to check for bugs in different environment behaviors, the oracle would need to be called with several seeds. Here, we show that many seed-bugs can be found even when only trying a single seed for each s_i.

The diversity filter serves (similarly as in software testing) for higher confidence in π's ability to avoid failure, based on broad tests. Diversity of program inputs is typically facilitated by only adding inputs to the pool if they increase some form of code coverage (e.g., statement, branch, or path coverage). Recently, several coverage criteria for NNs have emerged, such as neuron coverage [35] in the context of NN robustness testing. In π-fuzz, we take inspiration from a coverage criterion [33] based on the Euclidean distance between activation vectors for a given NN layer. Here, we apply this criterion to states – i.e., the NN input vectors – instead.

The RANDOMWALK procedure conducts random walks in the usual manner, with one noteworthy design decision. Rather than always choosing actions uniformly at random (line 23), the algorithm sometimes samples the policy under test instead (line 19; recall that $\hat{\pi}(s) \in \mathcal{D}(A)$ interprets the final layer of the NN policy as a probability distribution over actions). The parameter POL_PROB (line 17) controls the trade-off between these two choices. Sampling the policy makes sense when random actions do not tend to lead to interesting states, e.g., because states quickly become unsolvable. Indeed, as our empirical results show, this method yields advantages in all our case studies.

Regarding the parameters of Algorithm 2, in preliminary experiments we found that INC_PROB = 0.8 tends to work well across all of our case studies (for smaller values, exploration is insufficient), so we fix this parameter value. Similarly, we fix POL_PROB = 0.2 (for larger values, exploration is insufficient). For DIV_THRESH and WALK_LENGTH, good values depend on domain-specific aspects, i.e., typical scale of state diversity, typical run length, typical level of risk incurred by long random walks. We, hence, fix specific values for each domain, listed as part of our case study descriptions in the next section. For POL_PROB and DIV_THRESH, interesting algorithm performance differences arise from setting them to 0 vs. > 0, so we evaluate these settings in our experiments.

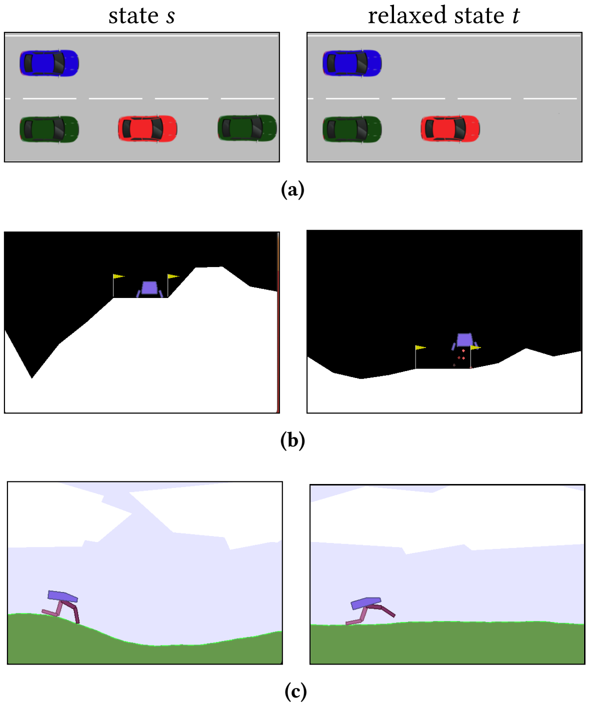
8 CASE STUDIES

We apply our π-fuzz framework to three case studies, called Highway, LunarLander, and BipedalWalker. Illustrations of their environments are shown in Figure 2. LunarLander and BipedalWalker are popular Gym [7] environments specialized for continuous control. We developed Highway as a new benchmark that simulates a simplified autonomous-driving task, navigating a highway through speed and lane changes in a way that avoids collisions with traffic. In all these case studies, the failure condition ϕ is given in terms of a specific environment state in which the agent ends up when it crashes into an obstacle. All agents were trained on a Debian 10 machine with 768 GB of memory, 32 CPUs (Intel(R) Xeon(R) Gold 6134M), and 2 GPUs (V100 Nvidia Tesla with 32 GB of memory).

We next describe each case study, including the domain itself, the algorithms and parameters we used for learning the policy π, the metamorphic operations for the oracle, and the domain-specific settings of DIV_THRESH and WALK_LENGTH.

8.1 Highway

The Highway domain consists of a two-lane, finite-length street. The left lane is for *speed maniacs*, who are relatively fast, and the right lane is for *safety freaks*, who are slow. Neither of these actors may change lanes, and their speed is constant. The agent appears at the beginning of the street, and the task is to reach the end without crashing into other cars. There are five discrete actions: switch lane to right or left, speed up, slow down, noop. Other cars may enter or leave the highway stochastically while the agent is moving. In case of a crash, the game ends immediately with a reward of −100 points. Reaching the end of the street is rewarded with +100 points. The discount factor is $\gamma = 0.95$, incentivizing the agent to drive fast. Given the road length, the best-case achievable reward is ca. 30.



(a)

(b)

(c)

Figure 2: Illustrations of domains and relaxations (as per Definition 2) used in our evaluation: (a) Highway, (b) LunarLander, and (c) BipedalWalker.

Policy training. We train the agent using our own implementation of DQN [32]. It is well trained, achieving an average reward of 21 points (after 20000 training episodes).

Metamorphic operations. Relaxed states are generated by removing a random car ahead of the agent, thus reducing the chance of crashing (see Figure 2a). Conversely, unrelaxed states are generated by adding a car at a safe distance from the agent (not leading to unavoidable crashes).

DIV_THRESH *and* WALK_LENGTH. We set DIV_THRESH to 3.6 to capture the typical scale of diversity in this domain, which we gauged by inspecting the visual differences between states. We set WALK_LENGTH to 3 to balance the typical risk of random walks, which quickly lead to crashes in this domain.

8.2 LunarLander

The LunarLander domain consists of an uneven lunar surface and a lander with two legs. The lander appears at the top of the environment with a random velocity vector, and the task is to land it on its legs – if the body touches the surface, the lander crashes. There are four discrete actions: firing the bottom engine, the left-hand side engine, the right-hand side engine, noop. The effect of firing an engine is stochastic, following a probability distribution over the yielded force. Touching a leg to the ground yields reward +100, and touching the body to the ground yields reward −100. There is no discount factor, and the best-case reward is over 200.

Policy training. We train the agent using PPO [37] implemented in the SB3 library [36]. Our agent is well trained, and achieves an average reward of 205 points (after 1 million training episodes).

Metamorphic operations. Relaxed states are generated by decreasing the height of the surface, giving the lander more time to land (see Figure 2b). Conversely, we generate unrelaxed states by increasing the surface height up to a safe distance.

`DIV_THRESH` *and* `WALK_LENGTH`. We set `DIV_THRESH` to 0.65 and `WALK_LENGTH` to 25.

8.3 BipedalWalker

In the BipedalWalker domain, a bipedal robot moves along a finite-length terrain that has a rough surface. The robot's task is to move forward until the end of the terrain. The action space is continuous, with actions being defined by a 4-tuple of numbers $x_i \in [0.0, 1.0]$. Each x_i specifies the force applied to one of the joints of the robot. The actions are deterministic; the only stochastic elements in this domain are the terrain (surface) shape and the initial forces in the robot's joints. The best-case achievable reward is +300 collected when reaching the end of the terrain, plus small positive rewards that can be collected beforehand. If the robot falls, it receives −100 points, and the game ends immediately. Again, there is no discount factor.

Policy training. We use the PPO algorithm from SB3 for training. Our agent achieves an average reward of 302 points (after 1 million training episodes).

Metamorphic operations. As smooth surfaces are easier to navigate for the walker, relaxed states are generated by making the terrain smoother (see Figure 2c), whereas unrelaxed states are generated by making the terrain rougher.

`DIV_THRESH` *and* `WALK_LENGTH`. We set `DIV_THRESH` to 2.0 and `WALK_LENGTH` to 25.

Overall, our case studies explore moving obstacles under simple action dynamics (Highway), fixed obstacles under complex action dynamics (LunarLander), and keeping balance with continuous motor control (BipedalWalker). They, thus, cover a broad range of applications featuring obstacle avoidance, which is a ubiquitous problem in neurally controlled systems.

9 EXPERIMENTS

Our primary evaluation concerns bug-finding capability, i.e., the number of (seed-)bugs correctly identified by different oracles. We, furthermore, analyze the impact of the `POL_PROB` and `DIV_THRESH` algorithm parameters, and we provide data on runtime to gauge the practical effort needed. In what follows, we first introduce the oracles we compare, then focus on these three evaluations in turn.

To account for the randomness in our algorithms and game environments, we run each experiment 8 times and report statistics over these runs below. Each run was performed on a Debian 10 machine with 1.5 TB of memory and 96 CPUs (Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz). For each run, we used a time limit of 24 hours.

9.1 Competing Oracles

To provide a comprehensive evaluation, we consider not only our metamorphic oracles, but eight oracles in total:

MMBug, MMSeedBugBasic, MMSeedBugExt These are the oracles from Algorithm 1. (MM stands for metamorphic.)

FailureSeedBug This oracle just flags s_i as a bug if $E^r.\sigma[\pi, s_i]$ fails.
This is a trivial baseline that does not check avoidability.

RuleSeedBug This oracle uses the aforementioned "change nothing" rule (see Section 2), reporting pool state s_i as a seed-bug if there is an unrelaxed state t_i, $R(t_i, s_i)$, such that $E^r.\sigma[\pi, t_i]$ succeeds and $\pi(s_i) \neq \pi(t_i)$. The rationale is that what works for t_i also works for s_i, so the policy should not change.
As $\pi(s_i) = \pi(t_i)$ is possible but not necessary, this oracle may incorrectly classify s_i as a seed-bug. Here, we report only the true positives, measuring the oracle's ability to identify true bugs (which as we shall see is lacking).

PerfectBug and PerfectSeedBug These oracles provide exact measuring lines. PerfectSeedBug explores all possible trajectories from a given state with the given random seed, and detects a bug only if there exists a winning trajectory but the policy fails from this state. PerfectBug simply measures $P_\phi(\pi, s)$ and $P^*_\phi(s)$, and detects a bug if $P_\phi(\pi, s) > P^*_\phi(s)$. Computing these oracles is tractable only for Highway, so we report data only for this case study.

MMSeedBug2Bug This oracle first calls MMSeedBugBasic. If MMSeedBugBasic flags s_i as a bug due to unrelaxed state t_i, then MMSeedBug2Bug flags s_i as a bug if additionally $P_\phi(\pi, t_i) < P_\phi(\pi, s_i)$.
Such seed-bug filtering speeds up bug-finding as we will see. Further, this oracle evaluates how many seed-bugs found by MMSeedBugBasic correspond to bugs.

In MMBug and MMSeedBug2Bug, we evaluate P_ϕ by running the policy 30 times. Based on limited experiments, this is reasonable in our case studies; using statistical methods to compute P_ϕ up to a confidence bound is future work.

Prior to considering the empirical data for these oracles, note the following guaranteed relations between the sets of states (or state/seed pairs) they identify as bugs:

RuleSeedBug ⊆ MMSeedBugBasic The true positives of the RuleSeedBug oracle are dominated by those of MMSeedBugBasic, because if $R(t_i, s_i)$ and $E^r.\sigma[\pi, t_i]$ succeeds, then s_i is a bug iff $E^r.\sigma[\pi, s_i]$ fails – which is precisely what MMSeedBugBasic is checking.

MMSeedBugBasic ⊆ PerfectSeedBug, MMBug ⊆ PerfectBug By Proposition 3.

PerfectSeedBug ⊆ FailureSeedBug FailureSeedBug catches all seed-bugs but may incorrectly flag non-bugs.

MMSeedBug2Bug $\subseteq$ **MMSeedBugBasic, MMSeedBug2Bug** $\subseteq$ **MMBug** By construction.

MMSeedBug2Bug = **MMSeedBugBasic** = **MMBug** This relation holds on deterministic domains where policy runs are unique.

The MMSeedBugExt oracle is incomparable to the others as it is the only one that attempts to find additional (seed-)bugs, beyond the pool states s_i.

9.2 Results: Oracle Capability

Figure 3 shows our evaluation of oracle bug-finding capability. We fix the default version of the fuzzer here (using the parameter settings as previously specified). For each case study, we plot pool size on the x-axis as the testing progresses, and we show how many bugs were reported by each oracle on the y-axis; dark lines denote mean values and shaded areas standard deviation.

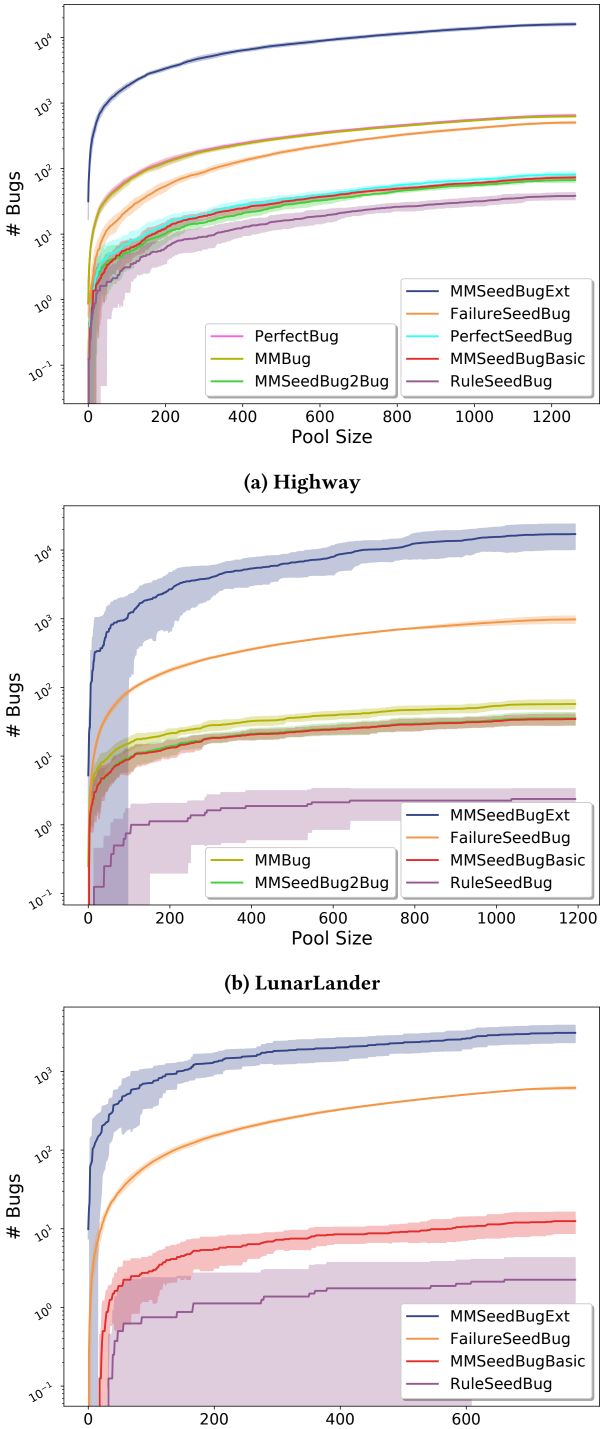
Consider first Figure 3a about Highway, where exact measuring lines by perfect oracles are available. These measuring lines attest to the strength of our metamorphic oracles in this domain: MMBug is close to PerfectBug, and MMSeedBugBasic is close to PerfectSeedBug. The average numbers of (seed-)bugs identified at the end of testing are 631.6 for MMBug, 650.1 for PerfectBug, 73.5 for MMSeedBugBasic, and 82.0 for PerfectSeedBug. Among the seed-bug oracles, FailureSeedBug reports many false positives, RuleSeedBug lags behind MMSeedBugBasic (38.5 at the end), and MMSeedBugExt finds a large number of additional bugs (~ 16000).

The gap between the seed-bug and bug oracles is large here. This is because the former ignore pool states that are solved by the policy under the one random seed chosen by the fuzzer, whereas the latter consider multiple seeds and identify many bugs for the same pool states. Accordingly, while MMSeedBug2Bug is close to MMSeedBugBasic showing that most seed-bugs we identify are bugs, MMSeedBug2Bug lags far behind MMBug.

Consider now LunarLander and BipedalWalker in Figure 3b and 3c. MMSeedBugBasic vastly outperforms RuleSeedBug. FailureSeedBug is far above that, but *at least* 50% of these failures are unavoidable. We calculate this number by implementing a limited-budget depth-first search, which examines all possible trajectories from a given state within a 15-minute time budget. We run this search on the pool states for which the policy fails, and we report the states for which a crash is unavoidable. There is such a large proportion of states where a crash is unavoidable because the LunarLander agent can easily get out of control with random actions and recovering becomes impossible. MMSeedBugExt finds many additional bugs as before (~ 17000).

In LunarLander, MMBug is only slightly above MMSeedBugBasic (in difference to Highway); MMSeedBug2Bug and MMSeedBugBasic are so close to each other that the two plots cannot be distinguished (99% of the seed-bugs reported by MMSeedBugBasic are bugs here). In BipedalWalker, the three plots necessarily coincide as the environment is deterministic. For this reason, we do not plot these lines.

Overall, the results show that **metamorphic oracles are highly useful for identifying bugs in action policies**, and they are superior to the rule-based and failure-based alternatives we evaluate. In the two domains where we are able to check (one with limited



(a) Highway

(b) LunarLander

(c) BipedalWalker

Figure 3: Evaluation of oracles: number of (unique) bugs as a function of pool size during the testing process. MMSeedBug2Bug and MMBug are not included in BipedalWalker as it is a deterministic environment, so these oracles coincide with MMSeedBugBasic.

Table 1: Evaluation of fuzzer configurations: number and diversity of bugs at the end of the testing process, using the MMSeedBugBasic oracle.

Setting / Domain	DIV_THRESH>0							
	POL_PROB=0.2				POL_PROB=0			
	# Bugs	Distance (L_2)			# Bugs	Distance (L_2)		
		Min	Max	Avg		Min	Max	Avg
Highway	73.5	3.6	59.8	12.1	50.5	3.6	62.6	13.6
LunarLander	34.6	0.7	3.5	1.6	15.3	0.7	3.1	1.4
BipedalWalker	13.3	2.0	6.3	3.8	13.0	2.0	6.3	3.6

Setting / Domain	DIV_THRESH=0							
	POL_PROB=0.2				POL_PROB=0			
	# Bugs	Distance (L_2)			# Bugs	Distance (L_2)		
		Min	Max	Avg		Min	Max	Avg
Highway	116.5	1.1	42.3	10.5	90.7	1.1	37.9	9.7
LunarLander	261.0	0>	2.7	1.0	167.2	0>	2.7	1.0
BipedalWalker	14.5	1.0	4.2	2.7	13.7	1.4	4.2	2.4

Table 2: Average runtime (in seconds). Left: Oracle runtime per state for MMSeedBugBasic, MMBug, and MMSeedBug2Bug. Right: Fuzzer runtime required to add one more state to the pool.

Domain	Oracle			Fuzzer			
				DIV_THRESH>0		DIV_THRESH=0	
	MMSBB	MMB	MMSB2B	POL_PROB=0.2	POL_PROB=0	POL_PROB=0.2	POL_PROB=0
Highway	0.4	10.1	0.6	73.0	96.9	0.001	0.003
LunarLander	0.6	10.1	1.4	82.4	158.0	0.004	0.004
BipedalWalker	6.1	n/a	n/a	119.6	159.0	0.008	0.010

budget), the oracles are close to perfect. Moreover, **seed-bug detection is a practical proxy for bug detection** in the sense that most seed-bugs detected by MMSeedBugBasic are bugs. Especially, **MMSeedBugExt is extremely effective in identifying bugs in diverse states**.

9.3 Results: Fuzzer Configurations

For our evaluation of fuzzer configurations – specifically, algorithm parameters POL_PROB and DIV_THRESH, which are the most interesting as discussed – see Table 1.

First, consider the impact of POL_PROB, controlling whether or not the policy under test is used to (partially) inform the random walks in the fuzzer. This is intended to improve the bug-finding capability in domains where purely random walks incur too many unavoidable failures. This effect is observed across domains, but it is especially noticeable in LunarLander, where a non-zero vs. a zero POL_PROB results in finding significantly more bugs for each setting of DIV_THRESH. (Note that value 0> in the table was 0.029 for POL_PROB=0.2 and 0.026 for POL_PROB=0 in our experiments.) A clear added value of finding more bugs is in using them as part of a targeted re-training process; as the amount of training data is important, so is the number of bugs.

On the other hand, a non-zero diversity threshold for adding states to the pool (DIV_THRESH) reduces the number of bugs found in LunarLander by up to an order of magnitude, with smaller reductions in Highway and BipedalWalker. This is because some detected bugs are not added to the pool, and there is a computational overhead associated with checking for diversity. The desired effect of increasing bug diversity is achieved though; all minimum, maximum, and average distance values among bug states are higher for a non-zero vs. a zero threshold. In general, diversity is important in gaining confidence that a policy is correct. Moreover, more diverse bugs could be more effective for re-training as they may point out different policy shortcomings. However, the gain in diversity requires more runtime, as we also show next.

9.4 Results: Fuzzer and Oracle Runtime

Finally, consider the runtime data in Table 2. We evaluate the MMSeedBugBasic and MMBug oracles to assess the effort required for identifying seed-bugs vs. bugs. We also include MMSeedBug2Bug to assess the speed-up gained from using seed-bug detection as a filter. As expected, seed-bug detection is much faster than bug detection, and seed-bug filtering gets rid of much of the overhead (at the risk of missing bugs, cf. above). Note that we do not evaluate MMBug and MMSeedBug2Bug for BipedalWalker as it is deterministic; these oracles coincide with MMSeedBugBasic.

Regarding fuzzer parameters, with DIV_THRESH>0, finding a new state for the pool takes 4–5 orders of magnitude more time (on average; at the start of testing, the overhead is smaller). Given that its only advantage is bug diversity (cf. Table 1), whether or not that switch should be activated depends on the application.

9.5 Threats to Validity

We identify the following threats to the validity of our experiments.

Games. The detected policy bugs, of course, depend on our selection of games. However, apart from the Highway benchmark that we developed, we show the effectiveness and generality of our approach by additionally applying it to two popular, off-the-shelf Gym environments. Further, we consider both deterministic and stochastic games.

Agents. The detected bugs also depend on the quality of the agents we train. However, all our agents are well trained as we describe in Section 8.

Metamorphic operations. The effectiveness of our approach is affected by the metamorphic operations that we define (see Section 8). However, as described earlier, relaxing obstacles is easy, hardly requiring any domain knowledge. When adding obstacles, we do so carefully to prevent causing unavoidable crashes, e.g., by adding a car at a safe distance from the agent in Highway, or by increasing the surface height up to a safe distance in LunarLander.

10 CONCLUSION

Testing action policies for avoidable failures requires oracles that can effectively identify sub-optimal failure-avoiding abilities. We have shown that such oracles can be obtained from relaxations, by adapting ideas from metamorphic testing. Our experiments confirm the potential of this approach.

This work opens up an entire universe of exciting research on relaxation-based metamorphic oracles. Possibilities include the automated design of state relaxations and thus metamorphic oracles; intelligent methods to explore environment behaviors in a search for seed-bugs; fault localization trying to identify specific combinations of policy decisions leading to failures; as well as closing the loop with re-training by feeding bug states back into reinforcement learning, until testing yields sufficient confidence in the policy.

ACKNOWLEDGMENTS

We are grateful to the reviewers for their constructive feedback. This work was supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

REFERENCES

[1] [n. d.]. LibFuzzer—A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.

[2] [n. d.]. Technical "Whitepaper" for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[3] Takumi Akazaki, Shuang Liu, Yoriyuki Yamagata, Yihai Duan, and Jianye Hao. 2018. Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning. In *FM (LNCS, Vol. 10951)*. Springer, 456–465.

[4] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. 2018. Safe Reinforcement Learning via Shielding. In *AAAI*. AAAI, 2669–2678.

[5] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *NeurIPS*. 2499–2509.

[6] Blai Bonet and Hector Geffner. 2001. Planning as Heuristic Search. *AIJ* 129 (2001), 5–33. Issue 1–2.

[7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* abs/1606.01540 (2016).

[8] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases.* Technical Report HKUST–CS98–01. HKUST.

[9] Anthony Corso, Robert J. Moss, Mark Koren, Ritchie Lee, and Mykel J. Kochenderfer. 2021. A Survey of Algorithms for Black-Box Safety Validation. *JAIR* 72 (2021), 377–428.

[10] Yao Deng, Xi Zheng, Tianyi Zhang, Guannan Lou, Huai Liu, and Miryung Kim. 2020. RMT: Rule-Based Metamorphic Testing for Autonomous Driving Models. *CoRR* abs/2012.10672 (2020).

[11] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. 2015. Red-Black Planning: A New Systematic Approach to Partial Delete Relaxation. *AIJ* 221 (2015), 73–114.

[12] Tommaso Dreossi, Thao Dang, Alexandre Donzé, James Kapinski, Xiaoqing Jin, and Jyotirmoy V. Deshmukh. 2015. Efficient Guiding Strategies for Testing of Temporal Properties of Hybrid Systems. In *NFM (LNCS, Vol. 9058)*. Springer, 127–142.

[13] Stefan Edelkamp. 2001. Planning with Pattern Databases. In *ECP*. AAAI, 13–24.

[14] Gidon Ernst, Sean Sedwards, Zhenya Zhang, and Ichiro Hasuo. 2019. Fast Falsification of Hybrid Systems Using Probabilistically Adaptive Input. In *QEST (LNCS, Vol. 11785)*. Springer, 165–181.

[15] Javier García and Fernando Fernández. 2015. A Comprehensive Survey on Safe Reinforcement Learning. *JMLR* 16 (2015), 1437–1480.

[16] Sankalp Garg, Aniket Bajpai, and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *ICAPS*. AAAI, 631–636.

[17] Raffaella Gentilini, Carla Piazza, and Alberto Policriti. 2003. From Bisimulation to Simulation: Coarsest Partition Problems. *J. Autom. Reason.* 31 (2003), 73–103. Issue 1.

[18] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Çakan. 2020. Importance-Driven Deep Learning System Testing. In *ICSE*. ACM, 322–323.

[19] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. 2018. Learning Generalized Reactive Policies Using Deep Neural Networks. In *ICAPS*. AAAI, 408–416.

[20] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-Invariant Testing for Machine Translation. In *ICSE*. ACM, 961–973.

[21] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *JACM* 61 (2014), 16:1–16:63. Issue 3.

[22] Jonathan Ho and Stefano Ermon. 2016. Generative Adversarial Imitation Learning. In *NeurIPS*. 4565–4573.

[23] Nathan Hunt, Nathan Fulton, Sara Magliacane, Trong Nghia Hoang, Subhro Das, and Armando Solar-Lezama. 2021. Verifiably Safe Exploration for End-To-End Reinforcement Learning. In *HSCC*. ACM, 14:1–14:11.

[24] Murugeswari Issakkimuthu, Alan Fern, and Prasad Tadepalli. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*. AAAI, 422–430.

[25] Rushang Karia and Siddharth Srivastava. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI*. AAAI, 8064–8073.

[26] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *ICSE*. IEEE Computer Society/ACM, 1039–1049.

[27] Mark Koren, Saud Alsaif, Ritchie Lee, and Mykel J. Kochenderfer. 2018. Adaptive Stress Testing for Autonomous Vehicles. In *IV*. IEEE Computer Society, 1–7.

[28] Ritchie Lee, Ole J. Mengshoel, Anshu Saksena, Ryan W. Gardner, Daniel Genin, Joshua Silbermann, Michael P. Owen, and Mykel J. Kochenderfer. 2020. Adaptive Stress Testing: Finding Likely Failure Events with Reinforcement Learning. *JAIR* 69 (2020), 1165–1201.

[29] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *ASE*. ACM, 120–131.

[30] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *CACM* 33 (1990), 32–44. Issue 12.

[31] Robin Milner. 1971. An Algebraic Definition of Simulation Between Programs. In *IJCAI*. 481–489.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-Level Control Through Deep Reinforcement Learning. *Nature* 518 (2015), 529–533. Issue 7540.

[33] Augustus Odena, Catherine Olsson, David Andersen, and Ian J. Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *ICML (PMLR, Vol. 97)*. PMLR, 4901–4911.

[34] Qi Pang, Yuanyuan Yuan, and Shuai Wang. 2021. MDPFuzzer: Finding Crash-Triggering State Sequences in Models Solving the Markov Decision Process. *CoRR* abs/2112.02807 (2021).

[35] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *SOSP*. ACM, 1–18.

[36] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. 2019. Stable Baselines3. https://github.com/DLR-RM/stable-baselines3.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017).

[38] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529 (2016), 484–489. Issue 7587.

[39] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2018. A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-Play. *Science* 362 (2018), 1140–1144. Issue 6419.

[40] Marcel Steinmetz, Timo P. Gros, Philippe Heim, Daniel Höller, and Jörg Hoffmann. 2021. Debugging a Policy: A Framework for Automatic Action Policy Testing. In *PRL@ICAPS*.

[41] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *ASE*. ACM, 109–119.

[42] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *ICSE*. ACM, 303–314.

[43] Sam Toyer, Sylvie Thiébaux, Felipe W. Trevizan, and Lexing Xie. 2020. ASNets: Deep Learning for Generalised Planning. *JAIR* 68 (2020), 1–68.

[44] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial Sample Detection for Deep Neural Network Through Model Mutation Testing. In *ICSE*. IEEE Computer Society/ACM, 1245–1256.

[45] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*. ACM, 132–142.

[46] Zhi Quan Zhou and Liqun Sun. 2019. Metamorphic Testing of Driverless Cars. *CACM* 62 (2019), 61–67. Issue 3.